As lazy as it can be

Rui CAMACHO

LIACC, *Rua do Campo Alegre, 823, 4150 Porto, Portugal*
FEUP, Rua Dr Roberto Frias, 4200-465 Porto, Portugal
e-mail: rcamacho@fe.up.pt
url: http://www.fe.up.pt/∼rcamacho

Abstract Inductive Logic Programming (ILP) is a promising technology for knowledge extraction applications. ILP has produced intelligible solutions for a wide variety of domains where it has been applied. The ILP lack of efficiency is, however, a major impediment for its scalability to applications requiring large amounts of data. In this paper we address important issues that must be solved to make ILP scalable to applications of knowledge extraction in large amounts of data. The issues include: efficiency and storage requirements. We propose and evaluate a set of techniques, globally called *lazy evaluation* of examples, to improve the efficiency of ILP systems. *Lazy evaluation* is essentially a way to avoid or postpone the evaluation of the generated hypotheses (coverage tests). To reduce the storage amount a representation schema called interval trees is proposed and evaluated.

All the techniques were evaluated using the IndLog ILP system and a set of ILP datasets referenced in the literature. The proposals lead to substantial efficiency improvements and memory savings and are generally applicable to any ILP system.

1 Introduction

Inductive Logic Programming (ILP) has achieved considerable success in a wide range of domains. It is recognized however that efficiency is a major obstacle to the use of ILP systems in applications requiring a large amounts of data. Relational Data Mining applications are an example where efficiency is an important issue. In this paper we address the problem of efficiency in ILP systems by proposing two techniques to improve it.

A typical ILP system carries out a search through an ordered hypothesis space. During the search hypothesis are generated and their quality estimated against the given examples. Improving the efficiency of such search procedure may be done by avoiding to generate useless hypothesis or/and improving the evaluation procedures.

Avoiding to generate useless hypotheses may be achieved with the specification of language bias limiting therefore the size of the search space ([1]). Another approach considers the study of refinement operators that allow to efficiently navigate through a hypothesis space ([2]).

The problem of efficient testing of candidate hypotheses has been tackled by the following techniques. Work of a stochastic nature (see [3, 4]). These reduce the evaluation effort at the cost of being correct only with high probability. A study on exact transformations of queries

when evaluating hypotheses may be found in [5] and [6]. In [5], the authors illustrated that query execution was a very high percentage of total running time.

Another line of research, as pointed out by Page [7], is the parallelization of an ILP system execution. It is also a very promising direction to overcome the efficiency bottleneck. Most of the techniques referred above are still applicable in a parallel execution setting and therefore substantial improvement on efficiency may be gained through the combination of the results of all of these lines of research.

In this paper we address two important issues of an ILP system: i) to avoid or reduce the computational cost in the evaluation of the hypotheses using the examples and; to reduce the amount of memory storage required. The first set of techniques is called *lazy evaluation* of examples and reduces considerably the amount of examples necessary to evaluate each hypothesis. The memory reduction is achieved by means of data structure called interval tree that is suggested to store the examples covered by individual hypotheses. Both techniques proposed may be adopted in any ILP system.

The structure of the rest of the paper is as follows. Section 2 presents a small introduction to ILP necessary to understand our proposals. In Section 3 we present the *lazy evaluation* of examples techniques. The data structure proposal is described in Section 4. In Section 5 we propose a set of efficiency improvements based on pruning properties of ILP systems. The experiments that empirically evaluate the proposals are presented in Section 6. The last section draws the conclusions.

2 ILP

This section briefly presents some concepts and terminology of Inductive Logic Programming but is not intended as an introduction to the field of ILP. For such introduction we refer to [8, 9, 10].

2.1 Problem

The objective of an ILP system is the induction of logic programs. As an input an ILP system receives a set of examples (divided in positive and negative examples) of the concept to learn, and sometimes some prior knowledge (or *background knowledge*). Both examples and background knowledge are usually represented as logic programs. An ILP system tries to produce a logic program where positive examples succeed and the negative examples fail.

From a logic perspective, the ILP problem can be defined as follows. Let E^+ be the set of positive examples, E^- the set of negative examples, $E = E^+ \cup E^-$, and B the background knowledge. In general, B, H, and E can be arbitrary logic programs. The aim of an ILP system is to find a set of hypotheses (also referred as a theory) H such that the following conditions hold:

- **Prior Satisfiability**: $B \wedge E^- \not\models \square$
- **Prior Necessity**: $B \not\models E^+$
- **Posterior Satisfiability**: $B \wedge E^- \wedge H \not\models \square$ (Consistency)

- **Posterior Sufficiency**: $B \wedge H \models E^+$ (Completeness)
- **Posterior necessity**: $B \wedge h_i \models e_1^+ \vee e_2^+ \vee \ldots \vee e_n^+ (\forall h_i \in H, e_j \in E^+)$

The sufficiency condition is sometimes named *completeness* with regard to positive evidence, and the posterior satisfiability is also known as *consistency* with the negative evidence. Posterior necessity states that each hypothesis h_i should not be vacuous.

The consistency condition is sometimes relaxed to allow hypotheses to be inconsistent with a small number of negative examples. This allows ILP systems to deal with noisy data (examples and background knowledge), i.e., sets of examples or background knowledge that contain some inaccuracies or other flaws.

2.2 ILP as a search problem

As explained before, the normal problem of ILP is to find a consistent and complete theory, i.e., a set of clauses that imply all given positive examples and is consistent with the given negative examples. Since it is not immediately obvious which set of clauses should be picked as the theory, a search among the permitted clauses is performed to find a set with the desired properties.

To find a satisfactory theory, an ILP system searches among a search space of the permitted clauses. Thus, learning can be seen as searching for a correct theory [11]. The states in the search space (designated as *hypothesis space*) are concept descriptions (hypothesis) and the goal is to find one or more states satisfying some quality criterion. For efficiency reasons the search space is structured by imposing a *generality order* upon the clauses. Such an order on clauses is usually denoted by $\succeq$. A clause C is said to be a generalization of D (dually: C is a specialization of D) if $C \succeq D$ holds. There are many generality orders, the most important are subsumption and logical implication. In both of these orders, the most general clause is the empty clause $\square$. The refinement operators [12] generalize or specialize hypothesis, thus generating more hypothesis.

The search can be done in two ways: specific-to-general [13] (or *bottom-up*); or general-to-specific [12, 14, 15, 16] (or *top-down*). In the generic-to-specific search the initial hypothesis is, usually, the more general hypothesis (i.e., $\square$). That hypothesis is then repeatedly specialized through the use of refinement operators in order to remove inconsistencies with the negative examples. In the specific-to-general search the examples, together with the background knowledge, are repeatedly generalized by applying refinement operators.

The hypotheses generated during the search are evaluated to determine their quality. A widely used approach to score a hypothesis is by measuring its accuracy (or coverage). The *accuracy* is the percentage of examples correctly classified by a hypothesis. The *coverture*, or coverage, of a hypothesis h is the number of positive (*positive cover*) and negative examples (*negative cover*) derivable from $B \wedge h$. The time needed to compute the coverage of a hypothesis depends, primarily on the cardinality of E^+ and E^-.

3 Lazy evaluation of examples

Language bias may be used to avoid the generation, and therefore, the evaluation of a significant number of hypotheses. However, once an hypothesis has been generated the problem then is how to evaluate it efficiently using the available data (examples and background

knowledge). This problem is specially critical if either the number of examples is large, like in Data Mining applications, or the evaluation of individual hypothesis (theorem proving effort) is hard [17]. The second problem has been addressed recently by means of techniques like query packs [18] or query transformations [5] and [6]. To handle the first situation, a probabilistic evaluation has been proposed [3] with the advantage of avoiding the use of all of the examples. This approach however prevents the system from having a correct measure of the hypothesis value.

We propose *lazy evaluation of examples* as a way to avoid unnecessary use of examples and therefore speed up the evaluation of each hypothesis. As the probabilistic approach we do not use all the examples to evaluate each hypothesis. Contrary to the probabilistic approach we get an exact count but only when it is absolutely necessary to do so. We can still profit from improvements due to query transformations and therefore combine the two technique for increasing speedup. We distinguish between lazy evaluation of positive examples, lazy evaluation of negative examples and positive evaluation avoidance. The techniques are now described.

Some systems like Progol [15], Aleph [19] or IndLog [16] rely on heuristics to guide the search during refinement[1]. If the search is complete then the role of the heuristic is to improve speed. The final hypothesis should be the same. Aleph, for example uses a *best first* algorithm in which the heuristic value of each clause is computed on the basis of the number of positive, negative examples covered and length of the clause. It is important for such heuristic to determine the exact number of examples covered. However, an hypothesis will only be accepted if it is consistent with the negative examples. We will call a clause that is inconsistent with the negative examples[2] a partial clause. In some applications an hypothesis is allowed to cover a small number of negative examples (the noise level). If a partial clause covers more than the allowed number of negative examples it must be specialised otherwise the search for further refinements of the partial clause terminates there. In this circumstances "lazy" evaluation of the negative examples is useful. When using the *lazy evaluation of negatives* we are only interesting in knowing if the hypothesis covers more than the allowed number of negative examples. We are not really interesting in knowing how much above the noise level the hypothesis is. Testing stops as soon as there are no more negative examples to be tested or the number of negative examples covered exceeds the allowed number for consistency (noise level). The noise level is quite often very close to zero and therefore the number of negative examples used in the test of each clause is very small. If the heuristic does not use the negative counting then this produces exactly the same results (clauses and accuracy) of the non-lazy approach but with a very significant speedup. It is also very common that the negative examples outnumber the positives. To use the *lazy evaluation* of negative examples the heuristic used should therefore not include negative cover. We may still use heuristics based on length and on positive cover.

IndLog also allows the positive cover to be evaluated lazily. A partial clause must be either specialised (if it covers more positives that the best consistent clause found so far) or is justifiably pruned away otherwise. When using lazy evaluation of positives it is relevant only to determine if an hypothesis covers more positives or not than the current best consistent

[1] The refinement step is the phase where the hypothesis space is searched.

[2] The hypothesis being too general.

hypothesis. We might then evaluate the positive examples just until we exceed the best cover so far. If the best cover is exceeded we retain the hypothesis (either accept it as final is consistent or refine it otherwise) if not we may justifiably discard it. Only when accepting a consistent hypothesis we need to evaluate its exact positive cover.

We may go a bit further and simply do not evaluate the positive cover at all of partial clauses. The advantage of the first version of lazy evaluation of positives is that we may discard hypotheses that are worse than the best consistent so far, while in the second version we may keep around some partial clauses with very poor positive cover. The advantage of the second version however is that for each hypothesis we only test it on the negatives until it covers at least the noise level. Generating hypotheses is very efficient and although we may generate more hypotheses we may still gain by the increase in speed of their evaluation process. This technique may be very useful in domains where the evaluation of each hypothesis is very time-consuming.

When performing lazy evaluation of positive and negative examples we may use a breadth-first search strategy for example. This is not a too bad choice if, like in most applications, one is looking for short clauses that are very close to the top of the subsumption lattice.

Lazy evaluation may be used together with other ILP speedup techniques to further increase the efficiency of the system. Lazy evaluation of negative examples may be used with the coverage caching technique as proposed by Cussens [20]. Coverage caching consists in storing the coverage lists permanently and reuse them whenever necessary, thus reducing the need to compute the coverage of a particular clause only once. Coverage lists reduces the effort in coverage computation at the cost of significantly increasing memory consumption.

Lazy evaluation of either negative or positive examples cannot be used when the technique called *lazy evaluation of literals* [21] is used. To compute the constant values all of the positive and negative coverage has to be computed exactly. It is also not applicable in data sets with positives only examples and the use of compression measure ([15]) or a user defined cost function like Aleph and IndLog might use ([21]).

4 *Interval trees*

Whenever an ILP system generates an hypothesis it has to evaluate it against the examples. If the number of examples is large or evaluating the hypothesis against each individual example is theorem proving expensive then it is critical to avoid unnecessary evaluation with all of the examples. For that purpose systems like IndLog, Aleph [19] or FORTE [22] maintain a list of the examples covered by each clause investigated. Whenever a new clause is evaluated the coverage list of the parent clause specifies which examples are used in the evaluation procedure. This approach speeds up the evaluation of new hypothesis but requires the coverage lists to be stored which may require large amounts of memory. Current versions of IndLog and Aleph store the coverage lists as interval lists like. In an interval list representation if a clause covers the following examples [1,2,3,7,8,10,10] it will be stored as [1,3,7,8,10,10] meaning that the hypothesis covers the examples from 1 to 3 and also examples 7 and 8 and 10 (a list of intervals). Although this representation as list of intervals is an improvement over the list of the individual examples number it still requires large amounts of storage for large datasets.

To reduce the amount of memory to store the coverage lists we propose the use of *interval trees*. An *interval tree* is a one dimension case of octree (three dimensions and quadtree for two dimensions). *Interval trees* as quadtrees or octrees are similar to k-d trees that are useful for N dimensional range queries problems. Basically an *interval tree* is a binary tree. The root node represents the all range interval (that must be a power of 2). If the hypothesis covers the all interval then we say the node (root) is black and is a leaf node. If the hypothesis does not cover any example in the interval then we say the node (root) is white and is again a leaf node. Otherwise we say the node (root) is gray and is a non leaf node with two sub-*interval-trees*. For the left subtree we apply recursively the same procedure stating that know the interval is from 1 to half of the original interval. For the right subtree we apply recursively the same procedure but with the interval from half the original to the max value.

One advantage of the interval-trees is that the intervals represented by each node are determined by the node's position in the tree and don't need to be explicitly stored. The only information explicitly stored is the color of each node. Since we need only three colors each node may be encoded with two bit. We may then store an interval-tree as a bit stream with two bit for each node.

Encoding each node with two bit one can store four nodes per byte. In our implementation we got a bit further and managed to store 5 nodes per byte. We represent the white nodes as zero, the black nodes as 1 and the gray nodes as 2. Each sequence of five nodes is converted in a number between 0 and 242 as

$$Char = \mathrm{N}_1 \times 3^4 + \mathrm{N}_2 \times 3^3 + \mathrm{N}_3 \times 3^2 + \mathrm{N}_4 \times 3 + \mathrm{N}_5$$

where the $\mathrm{N_i}$ represent the node's color ($0 < \mathrm{N_i} < 3$).

Consider that we have a dataset with 14 positive examples and that a certain clause covers the following ones: [1, 8, 13, 14] (from 1 to 8 and from 13 to 14). We first compute the smallest power of 2 that is equal or greater that 14 obtaining 16. To generate the interval-tree for this case we do the following. Since the clause does not cover all the examples between 1 and 16 the root node will be gray (*tree* = [2]). We divide the interval in two intervals 1 to 8 and 9 to 16. Interval 1 to 8 will be used to build the left sub-tree whereas the interval 9 to 16 will be used to construct the right sub-tree. Since the clause covers all examples between 1 and 8 we will make a black (leaf) node (*tree* = [21]). Since the clause does not cover all examples between 9 and 16 a gray node is made and two intervals generated one for each sub-tree (9 to 12 for the left one and 13 to 16 for the right one). The resulting tree will be (*tree* = [2120210]). This tree may be stored in 2 byte: $2 \times 3^4 + 1 \times 3^3 + 2 \times 3^2 + 0 \times 3 + 2$ = 209
$1 \times 3^4 + 0 \times 3^3 + 0 \times 3^2 + 0 \times 3 + 0$ = 81 whereas the range list needs 4 * 4 = 16 byte[3].

Our implementation of interval trees has a run-time overhead compared with the use of the interval lists. To avoid that overhead we may implement the trees in C as bit arrays or use a similar data structure called RL-Trees (**R**ange**L**ist-Tree) ([23]). The RL-Trees data structure is also an adaptation of a generic data structure called quadtree [24]. RL-trees do not have the run-time overhead in accessing the data items but require, generally, more memory space than interval trees.

[3] Assuming 4 byte to store an integer number

5 Further efficiency improvements

A typical ILP system implementing the Mode Directed Inverse Entailment [15] (MDIE) uses the individual examples as a seed to compute the most specific clause of the subsumption lattice to be traversed. Systems like Progol [15], Aleph [19], IndLog [16] and April [25] are based on MDIE. After traversing the subsumption lattice the *best* hypothesis is retained and the examples covered by it are removed. One of the remaining positive examples is then choose and the process iterates until all positive examples are covered. It is common in this systems that the removal of the covered examples is done only after using all the examples as seed or at least *sample size* of them. We argue that independent of the option taken there is valuable information collected during each traversal of the subsumption lattice that should be stored and used to speedup the search after the covered examples being removed. We propose that during each search the consistent clause with the large coverage difference with the best clause found should be retained. This information will establish minimum of positive cover for the next search after the examples removal. As an example consider that after each search through the subsumption lattice the examples covered are removed. Consider further that the system finds a consistent clause that covers less positive examples than the best clause found but covers N positive examples that are not covered by the best clause. After removing the examples covered by the best clause and when the next cycle begins we may establish that hypothesis covering less than N should be justifiably discarded and therefore speeding up the search.

ILP systems like Progol, Aleph or IndLog use justifiable pruning. Whenever a clause covers less positive examples that the best consistent clause already found then the search space *below* that clause may be justifiably discarded since it contains specializations (clauses with equal or less coverage). The sooner a consistent clause is found the sooner the system begins pruning using coverage. The larger the number of the best clause found the large the number of clauses we may discard during the search. Our proposal is to fake *ab initio* the discover of a large coverage clause. That is we start the search assuming we already have a N positives cover hypothesis. If the system does not find a consistent clause with at least that coverage then we repeat the cycle with a small fake value. This approach avoids the generation of low coverage clauses.

6 The experiments

To empirically evaluate our proposals we run IndLog [16] on several well known datasets. The experiments aim at estimating the efficiency gains when adopting the lazy evaluation of examples. By efficiency gains we mean a reduction in the number of theorem proving calls when evaluating the individual hypothesis. Both these measures are machine and implementation independent. We have also measured the CPU time reductions to see how effective the speedup are in an actual implementation. For each dataset the induced clause(s) are the same with and without lazy evaluation. Therefore the accuracy of the induced theories does not change by adopting any of the proposed techniques.

Dataset	positive examples	negative examples	background predicates
amine uptake	343	343	31
carcinogenesis	162	136	38
choline	663	663	31
krk (small)	342	658	3
krk (large)	3240	6760	3
mesh	2272	223	29
multiplication	9	15	3
mutagenesis	114	57	18
proteins (alpha)	848	764	45
pyrimidines	1394	1394	244
susi	253	8979	14
triazines	17063	17063	14

Table 1: Characterisation of the datasets used in the experiments.

Settings

The IndLog V1.0 system was used in the experiments. IndLog is encoded in Yap Prolog [26] (version 4.21) and run on a PC[4] with Linux. The datasets used in the experiments are characterised in Table 1 and were downloaded (except one) from the Oxford[5] and York [6] Universities Machine Learning repositories. The *susi* dataset was downloaded from the Science University of Tokyo[7].

Lazy evaluation of examples

In the experiments the IndLog settings were such that all lazy evaluations for the same dataset produce the same final theory and construct the same number of hypothesis. For the lazy evaluation of negatives we measure the number of theorem proving calls used to evaluate the negative examples and also the CPU time needed. For the lazy evaluation of positives we measured the number of theorem proving calls used to evaluate the positive examples and also the CPU time needed. For the no positive evaluation form of lazy evaluation we measured the number of theorem proving calls used to evaluate both the negative and positive examples, the CPU time needed and the number of nodes constructed. The results are shown in Table 2.

The Table 2 shows the percentage of the measured values when compared with the run when not using any kind of lazy evaluation. Except for the small (number of examples) datasets there is significant gain in using the lazy evaluation technique.

interval trees memory savings

Table 3 shows the percentage of memory required to store the coverage lists using the *interval trees* compared with the "usual" encoding of the interval lists. The reduction in memory size are very significant. We should stress that the amount of storage used in IndLog to store the coverage lists is nearly 40% of the total memory used.

[4]With an Athlon thunderbird CPU at 1GHz with 1GB of RAM.
[5]URL: http://www.comlab.ox.ac.uk/oucl/groups/machlearn/
[6]URL:http://www.cs.york.ac.uk/mlg/index.html
[7]URL:http://www.ia.noda.sut.ac.jp/ilp

Dataset	lazy negs		lazy pos		no pos			
	negs calls (%)	cpu time (%)	pos calls (%)	cpu time (%)	nodes (%)	negs calls (%)	pos calls (%)	cpu time (%)
amine uptake	25	75	69	99	146	56	14	97
carcinogenesis	27	79	80	115	105	76	23	68
choline	27	74	54	92	100	44	13	46
mesh	86	99	93	99	105	125	77	81
multiplication	31	99	91	104	152	95	44	127
mutagenesis	40	99	66	101	100	48	10	98
pyrimidines	38	76	63	99	120	62	13	58
susi	54	64	-	-	-	-	-	-
triazines	8	56	39	77	148	55	26	48

Table 2: Percentage of theorem proving calls and CPU time savings when using lazy valuation of examples. Comparison is made with the values obtained when not using any lazy evaluation.

Dataset	memory usage (%)
amine uptake	23.7
carcinogenesis	17.5
choline	21.5
krk (small)	16.9
krk (large)	15.9
mesh	23.5
multiplication	39.5
mutagenesis	19.1
proteins (alpha)	19.2
pyrimidines	19.4
suramin	27.9
susi	15.2
triazines	16.7

Table 3: Percentage of memory used when storing the coverage lists as *interval trees* by comparison with storing it as lists of integer intervals.

7 Conclusions

We have addressed two important issues when designing an ILP system: speed efficiency and memory requirements. We proposed and evaluated techniques to improve the efficiency of ILP systems and to reduce the amount of memory storage required.

To improve speed of any ILP system we propose the lazy evaluation of the examples. As shown by the empirical results this set of technique may produce substantial improvements to an ILP system.

Our proposal for reducing the amount of memory is called *interval trees*. This data structure produced a very significant reduction in the amount of memory used to process the datasets of the empirical evaluation.

Acknowledgments

The work presented in this paper has been partially supported by Universidade do Porto, project APRIL (Project POSI/SRI/40749/2001), funds granted to *LIACC* through the *Programa de Financiamento Plurianual*, *Fundação para a Ciência e Tecnologia* and *Programa POSI*.

References

[1] Nédellec, C. and Rouveirol, C. and Adé, H. and Bergadano, F. and Tausend, B., Declarative Bias in ILP, ed. De Raedt, L., Advances in Inductive Logic Programming, IOS, 82-103, (1996).

[2] Completeness and Properness of Refinement Operators, Patrick van der Laag and Shan-Hwei Nienhuys-Cheng, Journal of Logic Programming, vol 34, Nr 3, 201–226, (1998).

[3] A study of two sampling methods for analysing large datasets with ILP, A. Srinivasan, Data Mining and Knowledge Discovery, vol 3, Nr 1, 95–123, (1999).

[4] Tractable induction and classification in first-order logic via stochastic matching, M. Sebag and C. Rouveirol, Proceedings of the 15th International Joint Conference on Artificial Intelligence, Morgan Kaufmann, 888–893, (1997).

[5] Santos Costa, Vítor and Srinivasan, Ashwin and Camacho, Rui, A note on two simple transformations for improving the efficiency of an ILP system, Proceedings of the 10th International Conference on Inductive Logic Programming, Springer-Verlag, LNAI, vol 1866, ed. J. Cussens and A. Frisch, ISBN 3-540-67795-X, 225–242, (2000).

[6] Vítor Costa and Ashwin Srinivasan and Rui Camacho and Hendrik Blockeel and Bart Demoen and and Gerda Janssens and Jan Struyf and Henk Vandecasteele and Wim Van Laer, Query Transformations for Improving the Efficiency of ILP Systems, Journal of Machine Learning Research, (2002) (to appear).

[7] Page, David, ILP: Just Do It, Proceedings of the 10th International Conference on Inductive Logic Programming, Springer-Verlag, LNAI, vol 1866, ed. J. Cussens and A. Frisch, ISBN 3-540-67795-X, 3–18, (2000).

[8] Muggleton, S. and De Raedt, L., Inductive Logic Programming: Theory and Methods, Journal of Logic Programming, vol 19/20, 629-679, (1994).

[9] Lavrač, N. and Džeroski, S., Inductive Logic Programming: Techniques and Applications, Ellis Horwood, (1994)

[10] Foundations of Inductive Logic Programming, Nienhuys-Cheng, S.-H. and de Wolf, R., LNAI, vol. 1228, ISBN 3-540-62927-0, Springer-Verlag, (1997).

[11] T. Mitchell, Generalization as search, Artificial Intelligence, (1982), vol 18, 203-226.

[12] Shapiro, E.Y., Algorithmic Program Debugging, The MIT Press, (1983).

[13] Muggleton, S. and Feng, C., Efficient induction of logic programs, Proceedings of the 1st Conference on Algorithmic Learning Theory, Ohmsma, Tokyo, Japan, 368–381, (1990).

[14] Quinlan, J.R. and Cameron-Jones, R.M., FOIL: A Midterm Report, Proceedings of the 6th European Conference on Machine Learning, ed. Brazdil, P., 3–20, Springer-Verlag, LNAI vol 667, (1993).

[15] Muggleton, S., Inverse Entailment and Progol, New Generation Computing, Special issue on Inductive Logic Programming, 245–286, Vol 13, Nr 3-4, Ohmsha, (1995).

[16] Inducing Models of Human Control Skills using Machine Learning Algorithms, R. Camacho, Department of Electrical Engineering and Computation, Universidade do Porto, (2000).

[17] M. Botta and A. Giordana and L. Saitta and M. Sebag, Relational learning: hard problems and phase transitions, 178–189, Proc. of the 6th Congress AI*IA, LNAI 1792, Springer-Verlag, (1999).

[18] Blockeel, Hendrik and Dehaspe, Luc and Demoen, Bart and Janssens, Gerda and Ramon, Jan and Vandecasteele, Henk, Executing Query Packs in ILP, Proceedings of the 10th International Conference on Inductive Logic Programming, Springer-Verlag, LNAI, vol 1866, ed. J. Cussens and A. Frisch, ISBN 3-540-67795-X, 60–77, (2000).

[19] A. Srinivasan, http://web.comlab.ox.ac.uk/oucl/research/areas/machlearn/Aleph/.

[20] James Cussens, Part-of-speech disambiguation using ILP, Oxford University Computing Laboratory, PRG-TR-25-96, (1996).

[21] A. Srinivasan and R.C. Camacho, Numerical reasoning with an ILP program capable of lazy evaluation and customised search, Journal of Logic Programming, vol 40, Nr 2-3, 185–214, (1999).

[22] B.L. Richards and R.J. Mooney, Refinement of first-order Horn-clause domain theories, Machine Learning, vol 19, Nr 2, 95–131, (1995).

[23] Fonseca, Nuno and Rocha, Ricardo and Camacho, Rui and Silva, Fernando, Efficient Data Structures for ILP, Proceedings of the 13th International Conference on Inductive Logic Programming, Springer-Verlag, (2003) (to appear).

[24] Hanan Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys (CSUR)*, 16(2):187–260, 1984.

[25] The April ILP system, Fonseca, Nuno, Department of Computer Science, University of Porto, May, (2003).

[26] Costa, V. and Damas, L. and Reis, R. and Azevedo, R., YAP Prolog User's Manual, Universidade do Porto, (1989).